Case study Machine Learning Engineer

Objective: Explain the goal of the system: to enhance the model's performance by incorporating new feedback data iteratively.

Data Collection:

Source: I obtained the data from an image.mat file from the link: https://www.uco.es/kdis/mllresources/#20NGDesc

Format: The format of the data file. .mat is a format used by MATLAB to store variables, including arrays and matrices.

Content: It is stored in the image.mat file. For instance, it contain images in matrix form, labels, or other metadata.

Data Preparation:

Loading the Data: I used scipy.io.loadmat() to load the .mat file into Python.

Preprocessing:

Reshaping: The image matrices were reshaped into a format suitable for training (e.g., converting them to NumPy arrays).

Normalization: Pixel values were normalized to fall between 0 and 1 to improve model performance.

Label Encoding: Labels were encoded to fit a multi-label classification problem.

Splitting the Dataset: The dataset was split into training and validation sets to evaluate model performance.

CODE:

```
import scipy.io

# Load the .mat file

mat_data = scipy.io.loadmat('/content/Image.mat')

# Check the keys to understand the structure

print(mat_data.keys())

# Inspect the 'data' and 'target'

print(type(mat_data['data']))

print(type(mat_data['target']))

# print out their shapes or a sample of the data

print(mat_data['data'].shape)

print(mat_data['target'].shape)
```

print(mat_data['data'][:5])

print(mat_data['target'][:5])

import numpy as np

 # Assuming 'data' is your feature matrix and 'target' is your label vector

X = np.array(mat_data['data'])

y = np.array(mat_data['target'])

 # Normalize the data

X = X / 255.0

Checking and Handling Missing Values:

data_shape = X.shape

print(data_shape)

 # Calculate the dimensions

num_blocks = 49

num_channels = 3

num_statistics = 2

 # Calculate the image dimension

original_features = num_blocks * num_channels * num_statistics

print(original_features)

 # Reshape the data to match the block structure

num_images = X.shape[0]

X_images = X.reshape((num_images, 7, 7, 3, 2))

 # Total elements per image

num_features_per_image = X.shape[1]

print(f"Features per image: {num_features_per_image}")

 # Calculate total number of elements for given dimensions

```
expected_features = 7 * 7 * 3 * 2
print(f"Expected features per image: {expected_features}")

# Check for missing values
print(np.isnan(X).sum())
print(np.isnan(y).sum())

# Handle missing values if any (e.g., imputation)
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')
X = imputer.fit_transform(X)
```

Feature Scaling:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Reshape and Verify Data:

```
num_images = X.shape[0]
image_size = (7, 7, 3, 2)
X_images = X.reshape((num_images, *image_size))
print(y.shape)
print(y)
# Convert one-hot encoded labels to class indices
y = np.argmax(y, axis=0)
print(y[:5])
print(np.sum(mat_data['target'], axis=0))
import scipy.io
import numpy as np
```

```
# Load data
mat_data = scipy.io.loadmat('/content/Image.mat')
target = mat_data['target']

# Inspect the original target values
print("Original target shape:", target.shape)
print("First few values of target:", target[:, :5])

y = np.argmax(target, axis=0) if target.ndim > 1 else target.flatten()
# Verify conversion
print("Converted y shape:", y.shape)
print("First few labels after conversion:", y[:5])
print("Original target values (first few columns):", target[:, :5])
unique_values = np.unique(target)
print("Unique values in target:", unique_values)
if target.ndim > 1:
    y = np.argmax(target, axis=0)
else:
    y = target.flatten()

# Verify conversion
print("Converted y values (first few):", y[:5])
unique, counts = np.unique(y, return_counts=True)
print("Class label distribution:", dict(zip(unique, counts)))
from sklearn.model_selection import train_test_split

X = mat_data['data']
y = np.argmax(mat_data['target'], axis=0)  # Ensuring y is in correct format
```

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

Model Training:

Model Type: I used Convolutional Neural Networks (CNNs) for image classification.

Architecture: The CNN consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers and CNNs are highly effective in image recognition tasks as they can capture spatial hierarchies in images.

CODE:

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

Model Definition and Compilation:

```
# Define the CNN model

model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),

    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),

    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),

    Flatten(),

    Dense(64, activation='relu'),

    Dense(5, activation='softmax')  # Assuming 5 classes

])

# Compile the model

model.compile(optimizer='adam',

              loss='sparse_categorical_crossentropy',

              metrics=['accuracy'])

print(X.shape)

total_features = X.size / X.shape[0]

print(f"Total features per image: {total_features}")

expected_features = 64 * 64 * 3
```

```
print(f"Expected features per image: {expected_features}")
print("Data shape:", X.shape)
```

Checking with Random Forest:

```
param_grid = {
    'n_estimators': [100],
    'max_depth': [10],
    'min_samples_split': [2]
}
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint
param_dist = {
    'n_estimators': randint(50, 200),
    'max_depth': randint(5, 20),
    'min_samples_split': randint(2, 10)
}
clf = RandomForestClassifier(random_state=42)
random_search = RandomizedSearchCV(clf, param_distributions=param_dist, n_iter=10, cv=5, n_jobs=-1, random_state=42)
random_search.fit(X_train, y_train)
print("Best Parameters:", random_search.best_params_)
best_model = random_search.best_estimator_
from sklearn.model_selection import GridSearchCV
X_train_small, _, y_train_small, _ = train_test_split(X_train, y_train, test_size=0.9, random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(clf, param_grid, cv=5, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
```

best_model = grid_search.best_estimator_

 # Evaluate the best model

y_pred = best_model.predict(X_test)

from sklearn.metrics import accuracy_score

print("Accuracy:", accuracy_score(y_test, y_pred))

Retraining with CNN:

from tensorflow.keras.utils import to_categorical

 # Convert labels to one-hot encoding

y_train_cat = to_categorical(y_train, num_classes=5)

y_test_cat = to_categorical(y_test, num_classes=5)

print(f"Shape of X: {X.shape}")

print(f"Shape of y: {y.shape}")

 # To print shape and sample data from the mat file

print("Keys in mat_data:", mat_data.keys())

print("Shape of data:", mat_data['data'].shape) # Shape of features

print("Shape of target:", mat_data['target'].shape) # Shape of labels

 # y is a 1D array of labels

```
if y.ndim > 1:
    y = np.argmax(y, axis=1)
print(f"Number of samples in X: {X.shape[0]}")
print(f"Number of samples in y: {y.shape[0]}")
y = y[:2000]
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
```

 # Convert labels to categorical for classification

y_cat = to_categorical(y)

```
# Build a Dense Neural Network model
model = Sequential([
    Dense(128, activation='relu', input_shape=(X.shape[1],)),
    Dense(64, activation='relu'),
    Dense(5, activation='softmax')  # Assuming 5 classes
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train_cat, epochs=10, batch_size=32, validation_split=0.2)
loss, accuracy = model.evaluate(X_test, y_test_cat)
print(f"Test accuracy: {accuracy:.4f}")

# If y_pred is a 1D array, no need to apply np.argmax
if len(y_pred.shape) == 1:
    y_pred_classes = y_pred
else:
    # If y_pred contains probabilities, convert them to class labels
    y_pred_classes = np.argmax(y_pred, axis=1)

# Check the shape and type of y_test and y_pred_classes
print(f"y_test shape: {y_test.shape}")
print(f"y_pred_classes shape: {y_pred_classes.shape}")

# Compare with true labels
from sklearn.metrics import accuracy_score, classification_report

# Ensure y_test is a 1D array of class labels (not one-hot encoded)
accuracy = accuracy_score(y_test, y_pred_classes)
```

```
report = classification_report(y_test, y_pred_classes)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:")
print(report)
```

Parameter Tuning:

Hyperparameter Tuning: I used KerasTuner for hyperparameter tuning to optimize key parameters like the number of units in hidden layers, learning rate, and dropout rate.

Best Units: 384

Best Learning Rate: 0.001

And optimizing these hyperparameters is crucial for improving model performance and avoiding overfitting or underfitting.

CODE:

```
!pip install keras-tuner
import tensorflow as tf
from tensorflow.keras import layers
import keras_tuner as kt
def build_model(hp):
    model = tf.keras.Sequential()
    model.add(layers.Dense(
        units=hp.Int('units', min_value=32, max_value=512, step=32),
        activation='relu',
        input_shape=(X.shape[1],)
    ))
    model.add(layers.Dense(5, activation='softmax'))
    model.compile(
        optimizer=tf.keras.optimizers.Adam(
            hp.Choice('learning_rate', values=[1e-2, 1e-3])
        ),
```

```
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
tuner = kt.Hyperband(
    build_model,
    objective='val_accuracy',
    max_epochs=10,
    hyperband_iterations=2,
    directory='my_dir',
    project_name='intro_to_kt'
)
tuner.search(
    X_train, y_train_cat,
    epochs=10,
    validation_data=(X_test, y_test_cat)
)
best_model = tuner.get_best_models(num_models=1)[0]
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"Best units: {best_hps.get('units')}")
print(f"Best learning rate: {best_hps.get('learning_rate')}")
```

Update the model with best hyperparameters:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Build a Dense Neural Network model with best hyperparameters
model = Sequential([
    Dense(384, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(634, activation='relu'),
    Dense(5, activation='softmax')  # Assuming 5 classes
```

])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),

loss='categorical_crossentropy',

metrics=['accuracy'])

Training the model:

history = model.fit(X_train, y_train_cat,

epochs=10,

batch_size=32,

validation_split=0.2)

Evaluate the Model:

Evaluate on test data

test_loss, test_accuracy = model.evaluate(X_test, y_test_cat)

print(f"Test Loss: {test_loss:.4f}")

print(f"Test Accuracy: {test_accuracy:.4f}")

Make Predictions

Predict on test data

y_pred_prob = model.predict(X_test)

y_pred_classes = np.argmax(y_pred_prob, axis=1)

Calculate Metrics

from sklearn.metrics import accuracy_score, classification_report

 # Calculate accuracy

accuracy = accuracy_score(np.argmax(y_test_cat, axis=1), y_pred_classes)

print(f"Accuracy: {accuracy:.2f}")

 # Classification report

report = classification_report(np.argmax(y_test_cat, axis=1), y_pred_classes)

print("Classification Report:")

print(report)

Model Improvement:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

 # Build a more complex Dense Neural Network model

```
model = Sequential([
    Dense(256, activation='relu', input_shape=(X.shape[1],)),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dense(5, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

 # Train the model with improved architecture

```
history = model.fit(X_train, y_train_cat, epochs=20, batch_size=32, validation_split=0.2)
```

Evaluate:

 # Evaluate the best model on the test data

```
test_loss, test_accuracy = best_model.evaluate(X_test, y_test_cat)
print(f"Test Accuracy: {test_accuracy:.2f}")
```

 # Make predictions

```
y_pred = best_model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
```

 # Classification report

```
from sklearn.metrics import classification_report
print("Classification Report:")
```

print(classification_report(np.argmax(y_test_cat, axis=1), y_pred_classes))

from sklearn.metrics import confusion_matrix

import seaborn as sns

import matplotlib.pyplot as plt

import numpy as np

 # Assuming y_pred_classes and y_test_classes are already defined

y_pred_classes = np.argmax(model.predict(X_test), axis=1)

y_test_classes = np.argmax(y_test_cat, axis=1)

 # Generate confusion matrix

cm = confusion_matrix(y_test_classes, y_pred_classes)

 # Define class names

class_names = ['Class 0', 'Class 1', 'Class 2', 'Class 3', 'Class 4']

 # Plot confusion matrix

plt.figure(figsize=(10,7))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

K-Fold cross-validation:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Input

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.utils import to_categorical

from sklearn.model_selection import train_test_split

```
import numpy as np
# Convert labels to categorical
y_cat = to_categorical(y, num_classes=5)  # Ensure y is categorical and has 5 classes
# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y_cat, test_size=0.2, random_state=42)

# Define a function to create and compile the model
def create_model(units=128, learning_rate=0.001):
    model = Sequential([
        Input(shape=(X.shape[1],)),  # Define the input shape here
        Dense(units, activation='relu'),
        Dense(64, activation='relu'),
        Dense(5, activation='softmax')  # Output layer with 5 classes
    ])
    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Create and train the model
model = create_model()
history = model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=0, validation_data=(X_val, y_val))

# Evaluate the model
val_loss, val_acc = model.evaluate(X_val, y_val)
print(f"Validation accuracy: {val_acc}")
from sklearn.model_selection import KFold
import numpy as np

# Convert labels to categorical
```

```
y_cat = to_categorical(y, num_classes=5)


# K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cross_val_scores = []

for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y_cat[train_index], y_cat[val_index]

    model = create_model()
    model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=0, validation_data=(X_val, y_val))

    val_loss, val_acc = model.evaluate(X_val, y_val)
    cross_val_scores.append(val_acc)

print(f"Cross-validation scores: {cross_val_scores}")
print(f"Mean cross-validation score: {np.mean(cross_val_scores)}")

from sklearn.metrics import classification_report

# Predict on validation data

y_pred_probs = model.predict(X_val)
y_pred_classes = np.argmax(y_pred_probs, axis=1)
y_true_classes = np.argmax(y_val, axis=1)

# Classification report
report = classification_report(y_true_classes, y_pred_classes)
```

```
print("Classification Report:")
print(report)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Convert probabilities to class labels
y_pred_classes = np.argmax(y_pred, axis=1)

# Save the model
model.save('my_model.h5')
from tensorflow.keras.models import load_model

# Load the model
model = load_model('my_model.h5')
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

def create_model(units=128, learning_rate=0.001):
    model = Sequential([
        Dense(units, activation='relu', input_shape=(X_train.shape[1],)),
        Dropout(0.5),  # Add dropout for regularization
        Dense(64, activation='relu'),
        Dense(5, activation='softmax')
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
model = create_model(units=256, learning_rate=0.0005)  # Adjust units and learning rate
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2)

from tensorflow.keras.models import load_model
from sklearn.metrics import accuracy_score, classification_report
import numpy as np

# Load the saved model
model = load_model('/content/my_model.h5')

# Evaluate on test data
test_loss, test_accuracy = model.evaluate(X_test, y_test_cat, verbose=0)
print(f"Test Accuracy: {test_accuracy:.2f}")
print(f"Test Loss: {test_loss:.2f}")

# Make predictions on test data
y_pred = model.predict(X_test)

# Convert predictions to class labels
y_pred_classes = np.argmax(y_pred, axis=1)

# Calculate accuracy and generate classification report
accuracy = accuracy_score(np.argmax(y_test_cat, axis=1), y_pred_classes)
report = classification_report(np.argmax(y_test_cat, axis=1), y_pred_classes)

print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:")
print(report)
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, classification_report
```

```
import seaborn as sns
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical

# Assuming `y` is not one-hot encoded, we need to one-hot encode `y_train` and `y_val` in each split
kf = KFold(n_splits=5, shuffle=True, random_state=42)
accuracies = []

for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]

    # One-hot encode the training and validation labels
    y_train_cat = to_categorical(y_train, num_classes=5)
    y_val_cat = to_categorical(y_val, num_classes=5)

    model = create_model()  # Rebuild and train the model
    model.fit(X_train, y_train_cat, epochs=10, batch_size=32, verbose=0, validation_data=(X_val, y_val_cat))
    val_loss, val_accuracy = model.evaluate(X_val, y_val_cat, verbose=0)
    accuracies.append(val_accuracy)

print(f"Cross-validation accuracies: {accuracies}")
print(f"Mean cross-validation accuracy: {np.mean(accuracies)}")

# Confusion matrix for the test set
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Assuming y_test is also one-hot encoded, convert it to class labels
```

```
y_true_classes = np.argmax(y_test_cat, axis=1)


# Compute the confusion matrix
cm = confusion_matrix(y_true_classes, y_pred_classes)

# Plot the confusion matrix
plt.figure(figsize=(10,7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=np.arange(5), yticklabels=np.arange(5))
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
import numpy as np

# Convert your labels to one-hot encoding
y_train_cat = to_categorical(y_train, num_classes=5)  # Adjust num_classes if needed
y_val_cat = to_categorical(y_val, num_classes=5)

def create_model(units=128, learning_rate=0.001):
    model = Sequential([
        Dense(units, activation='relu', input_shape=(X_train.shape[1],)),
        Dense(64, activation='relu'),
        Dense(5, activation='softmax')  # Assuming 5 classes
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                  loss='categorical_crossentropy',
```

```
                metrics=['accuracy'])
    return model

# Create and train the model
model = create_model()
history = model.fit(X_train, y_train_cat, epochs=10, batch_size=32, verbose=1, validation_data=(X_val, y_val_cat))

# Evaluate the model
val_loss, val_accuracy = model.evaluate(X_val, y_val_cat, verbose=0)
print(f'Validation Loss: {val_loss:.4f}')
print(f'Validation Accuracy: {val_accuracy:.4f}')

import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.figure(figsize=(12, 6))
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
```

```
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
from sklearn.metrics import classification_report

# Predict on validation data
y_pred = model.predict(X_val)
y_pred_classes = np.argmax(y_pred, axis=1)
y_val_classes = np.argmax(y_val_cat, axis=1)

# Print classification report
report = classification_report(y_val_classes, y_pred_classes)
print("Classification Report:")
print(report)

import pandas as pd
import numpy as np

# Assuming you have predictions and true labels
y_pred = model.predict(X_test)  # Model predictions
y_pred_classes = np.argmax(y_pred, axis=1)  # Convert probabilities to class labels

# Create a DataFrame with the predictions and true labels
results_df = pd.DataFrame({
    'True_Label': y_test,
    'Predicted_Label': y_pred_classes
})
# Save to CSV
results_df.to_csv('model_predictions.csv', index=False)
```

Continuous learning through the feedback data

```
import numpy as np
import os
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import tensorflow as tf

# Function to create the model
def create_model(units=128, learning_rate=0.001):
    model = Sequential([
        Dense(units, activation='relu', input_shape=(X_train.shape[1],)),
        Dense(64, activation='relu'),
        Dense(5, activation='softmax')
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Initialize and split data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
y_train_cat = to_categorical(y_train, num_classes=5)
y_val_cat = to_categorical(y_val, num_classes=5)

# Initial training and saving the model
model = create_model()
model.fit(X_train, y_train_cat, epochs=10, batch_size=32)
```

```
model.save("initial_model.h5")
```

 # Function to simulate feedback data

```
def simulate_feedback():
    num_samples = 100
    num_features = X_train.shape[1]
    num_classes = y_train_cat.shape[1]
    new_samples = np.random.rand(num_samples, num_features)
    true_labels = np.random.randint(0, num_classes, num_samples)
    true_labels_cat = to_categorical(true_labels, num_classes=num_classes)
    return {
        "new_samples": new_samples,
        "true_labels": true_labels_cat
    }
```

 # Function to save the model with versioning

```
def save_model_with_version(model, base_path="models/", version=1):
    if not os.path.exists(base_path):
        os.makedirs(base_path)
    model_path = os.path.join(base_path, f"model_v{version}.h5")
    model.save(model_path)
    print(f"Model saved as {model_path}")
    return version + 1
```

 # Function for continuous learning

```
def continuous_learning_system(feedback_data, model_version):
    global X_train, y_train_cat, X_val, y_val_cat
    X_train = np.concatenate((X_train, feedback_data['new_samples']), axis=0)
    y_train_cat = np.concatenate((y_train_cat, feedback_data['true_labels']), axis=0)
    model = create_model()
    model.fit(X_train, y_train_cat, epochs=10, batch_size=32, verbose=0, validation_data=(X_val, y_val_cat))
```

```
    model_version = save_model_with_version(model, version=model_version)
    return model_version

# Lists to store performance metrics over versions
version_history = []
accuracy_history = []
loss_history = []

def log_performance(version, accuracy, loss):
    version_history.append(version)
    accuracy_history.append(accuracy)
    loss_history.append(loss)

# Simulate 5 updates and log performance
model_version = 1
for _ in range(5):
    val_loss, val_accuracy = model.evaluate(X_val, y_val_cat)
    print(f"Model Version: {model_version}")
    print(f"Validation Loss: {val_loss:.2f}, Validation Accuracy: {val_accuracy:.2f}")
    log_performance(model_version, val_accuracy, val_loss)
    feedback_data = simulate_feedback()
    model_version = continuous_learning_system(feedback_data, model_version)
# Plot performance trends
plt.figure(figsize=(10, 5))
plt.plot(version_history, accuracy_history, label='Accuracy')
plt.plot(version_history, loss_history, label='Loss')
plt.xlabel('Model Version')
plt.ylabel('Performance')
plt.title('Model Performance Over Time')
plt.legend()
```

```
plt.show()
print("Version history:", version_history)
print("Accuracy history:", accuracy_history)
print("Loss history:", loss_history)
```